Assignment No: 01

Name of student:

Roll No:

Practical Batch:

Title of the Assignment: DFS and BFS using recursive algorithm

Problem statement: Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Objective:

- To understand the concept BFS and DFS search techniques.
- To implement BFS and DFS search techniques using recursive functions on undirected graph or tree.

Theory:

Depth–first search (DFS):

Depth–first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before backtracking.

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks.

The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the

nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Note: Show an example for solving for DFS. Take an undirected graph/tree and show the traversal.

Algorithm:
Create a recursive function that takes the index of the node and a visited array.
1. Mark the current node as visited and print the node.
2. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Complexity Analysis:
- **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:** O(V), since an extra visited array of size V is required.

Pseudocode of recursive BFS:

```
DFS(adjacent[][], source, visited[], key) {
   if(source == key) return true //We found the key
   visited[source] = True

   FOR node in adjacent[source]:
       IF visited[node] == False:
          DFS(adjacent, node, visited)
       END IF
   END FOR
   return false    // If it reaches here, then all nodes have been explored
                  //and we still havent found the key.
}
```

Breadth–first search (BFS):

Breadth–first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first before moving to the next-level neighbors.

Note: Show an example for solving for BFS. Take an undirected graph/tree and show the traversal.

Algorithm:
1. Start by putting any one of the graph's vertices at the back of a queue.

2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Complexity Analysis:

- **Time complexity:** The worst case breadth-first search has to consider all paths to all possible nodes the time complexity of breadth-first search is O(|E| + |V|) where |V| and |E| is the cardinality of set of vertices and edges respectively.
- **Space complexity**: The space complexity can also be expressed as O(|V|) where |V| is the cardinality of the set of vertices

Pseudocode of recursive BFS:

```
recursiveBFS(Graph graph, Queue q, boolean[] visited, int key){
   if (q.isEmpty())
      return "Not Found";

   // pop front node from queue and print it
   int v = q.poll();
   if(v==key) return "Found";

   // do for every neighbors of node v
   for ( Node u in graph.get(v))
   {
      if (!visited[u])
      {
         // mark it visited and push it into queue
         visited[u] = true;
         q.add(u);
      }
   }
   // recurse for other nodes
   recursiveBFS(graph, q, visited, key);
}

Queue q = new Queue();
q.add(s);
recursiveBFS(graph, q, visited, key);
```

Conclusion:

We have implemented BFS and DFS using recursive algorithm for undirected graph.

Oral questions:

1. Why do we prefer queues instead of other data structures while implementing BFS?
2. Why is time complexity more in the case of graph being represented as Adjacency Matrix?
3. What are the classifications of edges in a BFS graph?
4. What is the difference between DFS and BFS? When is DFS and BFS used?
5. Can BFS be used for finding shortest possible path?
6. Is BFS a complete algorithm?
7. Is BFS a optimal algorithm?
8. Why can we not implement DFS using Queues? Why do we prefer stacks instead of other data structures?
9. What are the classifications of edges in DFS of a graph?
10. Can DFS be used for finding shortest possible path?
11. Why can we not use DFS for finding shortest possible path?
12. Is DFS a complete algorithm?
13. Is DFS a optimal algorithm?
14. When is it best to use DFS?
15. What are the applications of BFS and DFS?